Optimistic Locking with Concurrency in Oracle.

As Database Administrators are required to scale their databases to handle the challenges of Web-based access, B2B and E-commerce, faster hardware and more resources may only be a part of the solution. Poor locking strategies can cripple even the most well resource-backed applications. Optimistic Locking is often over-looked by architects and developers but may prove particularly useful for high transaction load databases such as those accessible by web-based clients. The following article explores the use of the Optimistic Locking with an Oracle database, and presents some battle-tested solutions for introducing the approach into any Oracle-based development project.

The Problems of Pessimistic Locking.

Most Oracle developers are already familiar with pessimistic locking - the technique by which the data to be updated is locked in advance. This is achieved using the familiar `SELECT…FOR UPDATE` syntax. Once the data to be updated has been locked, the application can make the required changes, and then commit or rollback - during which the lock is automatically dropped.

If anyone else attempts to acquire a lock of the same data during this process, they will be forced to wait until the first transaction has completed.

This approach is called pessimistic because it assumes that another transaction might change the data between the read and the update. In order to prevent that change from happening - and the data inconsistency that would result - the read statement locks the data to prevent any other transaction from changing it.

Whilst very simple and safe, this approach has two major problems....

- The Lockout - An application user selects a record for update, and then leaves for lunch without finishing or aborting the transaction. All other users that need to update that record are forced to wait until the user returns and completes the transaction, or until the DBA kills the offending transaction and releases the lock.
- The Deadlock - Users A and B are both updating the database at the same time. User A locks a record and then attempt to acquire a lock held by user B - who is waiting to obtain a lock held by user A. Both transactions go into an infinite wait state - the so-called deadly embrace or deadlock.

If you think these scenarios are rare then just ask anyone who has administered a large Rdb database. Rdb only offers pessimistic locking, and even with well-designed applications, lockouts and deadlocks can be frequent.

Detailed lock tree hierarchies - through which developers are required to obtain locks in a pre-defined order can resolve many of these problems, but this requires the Database Architect to identify all the possible transactions a new system will be required to make and ensure that they are catered for in the lock tree, as well as ensuring that all members of the development team are disciplined in following the tree. Even if the Database Architect correctly identifies all the transactions and the Developers faithfully follow the lock tree, future releases of the software may add new transactions that cannot be easily integrated into the lock tree. Furthermore, GUI interfaces such as those used by web-based applications often offer users a variety of methods of navigating to the same result. Mapping this user-driven complexity into a rigid lock tree can prove difficult.

Modern servers are capable of supporting hundreds if not thousands of simultaneous transactions which only increases the potential problems posed by pessimistic locking. The Oracle Parallel Server, which can dramatically improve throughput for very large databases by leveraging the benefits of clustered hardware systems can also be crippled by excessive locking as the distributed lock manager tries to co-ordinate each pessimistic lock amongst all of the instances in the database.

Introducing Optimistic Locking.

Optimistic locking offers an elegant solution to the problems outlined above. Optimistic locking does not lock records when they are read, and proceeds on the assumption that the data being updated has not changed since the read. Since no locks are taken out during the read, it doesn't matter if the user goes to lunch after starting a transaction, and deadlocks are all but eliminated since users should never have to wait on each other's locks.

The Oracle database uses optimistic locking by default. Any command that begins with `UPDATE…SET` that is not preceded by a `SELECT…FOR UPDATE` is an example of optimistic locking.

However, whilst this is optimistic locking at work, it does not provide concurrency control. Concurrency control is the mechanism that ensures that the data being written back to the database is consistent with what was read from the database in the first place - that is that no other transaction has updated the data after it was read.

To demonstrate this, consider an example based on the familiar SCOTT/TIGER demonstration:

It's the end of the year and Scott has done well. King, his manager, decides to give him a well-earned $300 pay rise. Meanwhile Human Resources are also using the Employee system, giving everybody a 5% annual salary adjustment….

King reads Scott's salary data and gets distracted….

```
SQL> select SAL from EMP where EMPNO = 7788;
       SAL
----------
      3000
```

Human Resources read Scott's salary data, apply a 5% increase and commit the change:

```
SQL> select SAL from EMP where EMPNO = 7788;
       SAL
----------
      3000

SQL> update EMP set SAL = 3150 where EMPNO = 7788;
1 row updated.

SQL> commit;
Commit complete.
```

King computes Scott's new salary based on his obsolete data and updates the record:

```
SQL> update EMP set SAL = 3300 where EMPNO = 7788;
1 row updated.

SQL> commit;
Commit complete.
```

The change made by Human Resources has been wiped out:

```
SQL> select SAL from EMP where EMPNO = 7788;
       SAL
----------
      3300
```

Since the locking mechanism is optimistic, no locks are taken out during the database read. When King gets distracted and HR make their 5% adjustment, King has no way of knowing that his data has become obsolete. Unaware of what Human Resources has done, King calculates Scott's new salary on his original

(and now inconsistent) data producing a result of $3300 which he commits back to the database wiping out the change made by HR. This is what is known as a *"buried update"*. Instead of the $3450 a year Scott was supposed to be making he now makes $3300 - and later quits to find a better paid job designing databases.

Once again you may believe that this scenario is rare - but anyone who thinks the credit card gods smiled on them by not charging for that purchase several months ago have in fact probably witnessed a case of transaction concurrency failure. Similarly, anyone who has checked in for a flight and been issued the same boarding pass as someone else is probably the victim of a badly designed optimistic lock rather than an unscrupulous airline engaging in passenger seating foul play.

Whereas most of these transaction concurrency failures result in rather minor problems such as a free CD from that on-line store or a double booked seat, both of which can easily be rectified by a polite conversation with the relevant person, when you consider safety critical systems the risks of concurrency failure using this approach become unacceptable. What would be the consequences of an optimistic lock failure resulting in the same organ being issued to two critically ill patients, or two aircraft being instructed to land on the same runway at the same time?

Ideally what we need is a method of allowing optimistic locking whilst ensuring concurrency, and although the technique is well documented and understood, it is rarely used in business applications.

Ensuring Data Concurrency.

The method to ensure data consistency with the use of optimistic locking is quite simple. At it's most basic it involves reading a key value along with the record, and then writing that same key value back to the database when the record is updated. The database then modifies the key value on a successful write to ensure that any other current transactions that hold the same key value will not be able to modify the record. Should any other transaction attempt to update the record with the old (and now obsolete) key value, the database will recognize the key value as obsolete and reject the update.

In practice there are a number of different way of achieving this, but the most common is the use of a modification time-stamp. Let us consider again our Employee example from earlier:

King would read Scott's salary data from the database with a time-stamp, and then get distracted.

Human Resources would read the same record and add 5% to Scott's salary. HR would then write back the record with the same time-stamp that both they and King have read. The database would compare the time-stamps, see that they match, and accept the update. The database now updates the time-stamp to reflect the fact that the record has changed.

King would then return to his task of giving Scott his well-earned $300 pay rise and attempt to update the record with the original and now obsolete time-stamp. The database would compare the time-stamps and see that they no longer match. The database would reject the update with a concurrency failure and the well-written application would inform King that the record he is trying to update has been changed and must be re-read before any updates can be made.

King re-reads the record and applies the $300 pay rise to Scott's adjusted salary giving a final and correct figure of $3450

Note that it is the database that updates the time-stamp - not the application. The time-stamp in this context is merely a key that demonstrates to the database that the data being written is consistent with that already in the record.

Suggestions for Implementation.

The Oracle database, with it's row-level triggers and sophisticated PL/SQL programming language provides everything we need to implement concurrency controls for Database Architects wishing to use optimistic locking. Indeed, there are doubtless many ways of achieving concurrency control with an optimistic lock, but in this article I present one method that has been refined over the development cycles of several large projects. The reader is invited to take this suggestion and modify it to their own needs.

However, before describing the solution I would like to cover some of the lessons that have helped to shape it's evolution:

- The concurrency should always be enforced at the trigger level. I have seen several attempts to enforce concurrency using a PL/SQL interface, and although it can work, is easily defeated by the casual user with access to a SQL prompt and a rudimentary understanding of SQL.
- The concurrency key used to guarantee consistency should never be called "Timestamp" - even if it is a date/time-stamp. Calling the concurrency key "Datestamp" or "Timestamp" is inviting the Development team to misinterpret the meaning of the data held in the column. I have seen concurrency key time-stamps appearing on management reports with all manner of imagined meanings.
- Using a date/time-stamp as the concurrency key is further discouraged as it only offers a one-second resolution for the lock. Although today it may seem unlikely that two transactions could read the same record, make different changes and both write back in the same second, the rapid pace of development in the information sector and the increasing use of B2B systems mean that the likelihood of this scenario will only increase over time.

- Using an integer for the concurrency key makes interactive SQL much simpler, and if combined with the `get_time` function within the `dbms_utility` package offers lock resolutions to 100th second.

- Use an offset when writing back the key. For example, if the key value read is *n*, then the application should write back *n+1*. This is for two reasons. Firstly, if the transaction making the update only had to specify the same key as was read, then not specifying the key at all would have the same effect, and so would require the update trigger to include code to check that the key was being updated. Secondly, even if the trigger did include code to ensure the key was being updated, many third-party transaction processors will attempt to out-smart the application developer and screen out updates that they consider redundant. Third-party transaction processors are becoming more commonplace with the advent of B2B communications and applications that interact with heterogeneous data-sources. When your well-behaved application dutifully attempts to write back exactly the same key value as it read, a third-party transaction processor may determine that update to be redundant and omit it from the update command that finally reaches the database, resulting in a de-facto concurrency failure. This is especially true of several of the current leading application servers on the market.

- When a concurrency failure is encountered abort the transaction immediately using PL/SQL's `raise_application_error` procedure. Do not attempt to handle the error any other way - to do so is to risk inconsistent data in your database.

Worked Example.

In this section we present a fully tested worked example that draws upon all of the lessons presented in the previous section and which is again based on the familiar SCOTT/TIGER demonstration. This example will work in all versions of Oracle from 7.1 onwards:

First we will add our concurrency key to the EMP table as an integer column called TCN (Transaction Control Number). We then set the initial value of the TCN for each row and mark the column as not- null.

```
SQL> alter table EMP add TCN integer;

Table altered.

SQL> update EMP set TCN = dbms_utility.get_time;

14 rows updated.

SQL> alter table EMP modify TCN not null;

Table altered.
```

We can now add the pre-insert and pre-update triggers that will enforce concurrency with our optimistic locking strategy. The pre-insert trigger simply sets the initial value of the TCN. Note that the initial value can be any arbitrary integer value, the only purpose at this stage is to comply with the not-null requirement, but using the result of the `get_time` function provides consistency across both triggers. The real work is done by the pre-update trigger that checks the value of the updated TCN against the one held in the database. If the two TCNs do not match (they are in fact offset by one) the transaction is rejected.

```
create or replace trigger EMP_BITRG
before insert on EMP for each row
begin
        /* NOTE - additional pre-insert code may go here */
        /* set the initial transaction control number */
        :new.TCN := dbms_utility.get_time;
end;

create or replace trigger EMP_BUTRG
before update on EMP for each row
begin
        /* test for concurrency failure */
        if( :new.TCN != :old.TCN+1 ) then
             raise_application_error( -20000, 'Concurrency Failure' );
        end if;

        /* NOTE - additional pre-update code may go here */
        /* update the transaction control number */
        :new.TCN := dbms_utility.get_time;
end;
```

Now let's re-run our example with King and Human Resources to see what happens:

King reads Scott's salary data and gets distracted….

```
SQL> select SAL, TCN from EMP where EMPNO = 7788;
       SAL        TCN
---------- ----------
      3000  761824625
```

Human Resources read Scott's salary data, apply a 5% increase and commit the change:

```
SQL> select SAL, TCN from EMP where EMPNO = 7788;
       SAL        TCN
---------- ----------
      3000  761824625

SQL> update EMP set SAL = 3150, TCN = 761824626 where EMPNO = 7788;
1 row updated.

SQL> commit;
Commit complete.
```

King computes Scott's new salary based on his obsolete data and attempts to update the record:

```
SQL> update EMP set SAL = 3300, TCN = 761824626 where EMPNO = 7788;
update EMP set SAL = 3300, TCN = 761824626 where EMPNO = 7788
       *
ERROR at line 1:
ORA-20000: Concurrency Failure
ORA-06512: at "SCOTT.EMP_BUTRG", line 4
ORA-04088: error during execution of trigger 'SCOTT.EMP_BUTRG'
```

King re-reads Scott's salary data and discovers the change:

```
SQL> select SAL, TCN from EMP where EMPNO = 7788;
       SAL        TCN
---------- ----------
      3150  761828854
```

King adjusts his calculations to incorporate the change:

```
SQL> update EMP set SAL = 3450, TCN = 761828855 where EMPNO = 7788;
1 row updated.
```

After King successfully applies his $300 pay rise the pre-update trigger in the database changes the TCN number again to ensure that Human Resources nor anyone else can update the database with an obsolete TCN.

Limitations and Suggestions for Future Enhancements.

The solutions and example shown above demonstrate how to integrate optimistic locking with concurrency control into an Oracle development project. However the Database Architect needs to understand that this is only part of the solution.

More challenging than the technology is overcoming resistance from seasoned development professionals who have been using the trusted `SELECT… FOR UPDATE` for all of their Oracle careers. These individuals may need to be convinced of the benefits of using optimistic and on large development projects their support will be crucial.

Also note that the solution as shown is still not fool-proof. Any transaction that updates a record and sets the TCN to TCN+1 will succeed. Indeed, if the update being made is a relative update (e.g. set X = X * 1.05) then defeating the concurrency control in this manner is acceptable. What is not acceptable is defeating the concurrency control to perform an absolute update (e.g. set X = 42) if the validity of that update would be undermined by X changing since the new absolute value was calculated.

The use of the `dbms_utility.get_time` function to set and update the TCN has proven reliable on several large projects, but others may decide on a different approach such as a straightforward unique sequence number.

As noted earlier, many third party transaction processors and application servers are already implementing optimistic locking with concurrency controls, and it is probably only a matter of time before Oracle Corp. integrates some form of concurrency control into the core of the database product.